# Using Provenance to Extract Semantic File Attributes

Daniel Margo
*Harvard University*

Robin Smogor
*Harvard University*

## Abstract

Rich, semantically descriptive file attributes are valuable in many contexts, such as semantic namespaces and desktop search. Descriptive attributes help users to find files placed in seemingly-arbitrary locations by different applications. However, extracting semantic attributes from file contents is nontrivial. An alternative is to examine file provenance: how and when files are used, and the agents that use them.

We study the extraction of semantic attributes from file provenance by applying data mining and machine learning techniques to file metadata. We show that provenance and other metadata predict semantic attributes such as file extensions. This complements previous work, which has shown that file extensions predict access patterns.

## 1 Introduction

Semantic attributes, which describe an object in human-readable terms, are useful to many applications. For example, iTunes represents a music collection as a semantic namespace in which songs are located by attributes such as album, artist, and genre. Desktop search engines such as Google Desktop Search also locate data semantically and benefit from descriptive attributes.

One of the fundamental challenges in semantic applications is the problem of extracting attributes. A semantic application is not a useful tool unless it has rich, accurate attributes to work with. Unfortunately, manual labeling is an arduous task (akin to assigning a file multiple directories) and is intractable for importing extant systems. This "labeling problem" has been the subject of research, but is far from solved. Recent projects extract acoustic features from music [1] and summaries and other features from text documents [7]. However, these systems are necessarily limited in that they must understand how to read and interpet the contents of each type of file. Furthermore, they treat files as individuals and do not consider *context*: historical and other relationships between files.

We propose that a great deal of information can be obtained by examining file provenance: how and when files are used, and the agents that use them. For example, a file that is always opened by an application and is contained in that application's directory is likely to be part of that application. Conversely, a file that is always opened by many applications but is not contained in any application's directory is likely to be a library. A file that is occasionally opened by an application and is not contained in that application's directory is likely to be content manipulated by that application. One can imagine a machine learning classification algorithm that matches file provenance to patterns such as "application component".

There is significant prior work on using contextual file metadata in desktop search. Shah et al. have used provenance to improve desktop search [5], and Soules and Ganger have researched attribute propagation using content similarity and temporal context [6]. Temporal context can be thought of as a coarse approximation of provenance, because objects that share a provenance relationship must be active in the same timespan. True provenance records contain more data at a finer granularity, but this consequently introduces novel challenges.

Classifying provenance is challenging because provenance data is large and multi-dimensional. Provenance is generally represented as a graph, and the size of a graph is quadratic in its number of nodes. Furthermore, each node and edge is labeled with metadata such as name and version. In contrast, a typical machine learning classifier can handle feature vectors on the order of tens or hundreds of features in length. Therefore, we must intelligently reduce the large graph to a few relevant features.

We use provenance collected by the Provenance-Aware Storage System v2 [4] to describe file history. Then, using a variety of clustering and machine learning algorithms, we classify files by their provenance and other metadata. Because this data is large, we ex-

| | | | | |
|---|---|---|---|---|
| De-version | Merge Names<br>Don't Merge | Provenance Graph<br>File Graph<br>Process Graph | Ancestors<br>Descendants | Node Count<br>Edge Count<br>Max Depth<br>Neighbors |

Table 1: The feature extraction pipeline. Each step is chosen from left to right.

plore different methods of distilling and extracting relevant features. We demonstrate that provenance and other metadata are predictive of extant semantic attributes, such as file extensions.

## 2 Design

Our high-level goal is to take the provenance of a set of files, and output semantically meaningful classifications. We approach this problem in three stages: collecting provenance, processing provenance, and feeding processed provenance to a machine learning classifier. The processing stage can be further broken down into a set of component techniques that assemble in various ways. In the following section, we broadly map out this design and its implications.

We collect provenance using PASSv2 [4], which captures provenance relationships at the Linux syscall interface. For example, PASSv2 captures and logs the source and destination of a processes's write to a file. The resulting output is a directed acyclic graph in which the process and file are nodes, and the write is an edge.

When a process writes to a file it has previously read, this creates a cyclic dependency that PASSv2 resolves by creating a new version of the file that implicitly depends on the old version. As a result, PASSv2 nodes do not correspond to files and processes, but rather historical instances of files and processes. This distinction is typical in provenance systems, and is important both when we process provenance and interpret the results.

### 2.1 Processing Provenance

In the processing stage, we reduce the large, singular provenance graph to a small number of per-file features. Graph features can be divided into two categories, extrinsic and intrinsic. Extrinsic features are attributes and labels of nodes and edges that are particular to a given class of graphs (in this case, PASSv2 graphs). Conversely, instrinsic features are structures that depend solely on graph topology. Extrinsic features are often semantically meaningful, but intrinsic features are more generalizable.

#### 2.1.1 Extrinsic Features

Probably the most significant extrinsic feature in PASSv2 (and many other provenance systems) is node versioning. Recall that, due to versioning, there is no one-to-one correspondence between files and nodes. Therefore, we must reconcile multiple version nodes when generating per-file features. One simple solution is to *de-version* the graph by merging nodes that refer to a single versioned object. Alternatively, we could process the versioned graph, generate per-node features, and reconcile them in post-processing (for example, by averaging versions, or discarding all but the most recent version).

De-versioning emphasizes the relationship between different versions of a file and reduces graph sparsity, but also discards topological information and introduces cycles and false dependencies. Conversely, post-processing lets us retain and better manage topological information, but does not address sparsity and introduces further decisions with regards to how the versions are reconciled. In our initial work we have found sparsity to be a challenge and the decision space to be large, so to date we have only operated on de-versioned graphs. We can further explore this concept and reduce sparsity by merging nodes with identical pathnames. This ensures one node per file, and captures relationships such as instances of a process or an application's temporary files, but can also introduce wholly false relationships. However, it is possible that small amounts of such noise will be "washed out" by the machine learner.

Other notable extrinsic features in PASSv2 include node types and edge timestamps. Node types are semantic labels such as "file", "process", etc. We can reduce a provenance graph to a file graph or a process graph by omitting all non-file or non-process objects, respectively. The former emphasizes dependencies between files, whereas the latter emphasizes workflow. These alternative representations of the graph are then interesting candidates for intrinsic feature analysis, as described below. We plan to incorporate edge timestamps into our processing pipeline in future work.

In addition, we collect features from file system metadata. Per-file features such as directory depth, last access time, etc. are readily available via a `stat` syscall. While these features are not provenance, they are consistent with our file content-ignorant approach.

#### 2.1.2 Intrinsic Features

A traditional technique to summarize intrinsic graph topology is topological clustering. Typical clustering al-

gorithms take a single graph as input and, using the local topology of each node, partition the graph. Unfortunately, partitioners perform poorly on provenance graphs because they are often sparse, and because the partitioner is ignorant of the properties of provenance graphs. In particular, partitioners are not aware that the ancestor and descendant subgraphs of a given node have special significance; they only consider local topology. Across-graph clusterers could compare these sets of subgraphs for similarity, but they can be slow on large graphs and also suffer from sparsity.

However, we can easily collect simple statistics about each file's ancestors and descendants. Features such as the number of nodes and edges, the maximum path length, and the file's immediate neighbors can be calculated for each file's ancestor and descendant subgraphs. Furthermore, these features can be collected on different representations of the graph: provenance, file, and process graphs, with merged or unmerged pathnames. Note that since our de-versioned graphs can contain cycles, we define the ancestor and descendant subgraphs of a file as the transitive closure of children-to-parent and parent-to-children traversals, respectively. The feature extraction pipeline (excluding `stat`) is summarized in table 1.

## 2.2 Machine Learning

The goal of the machine learning stage is to further reduce our per-file features into an intelligent attribute prediction for each file. However, we would also like to understand the learners reasoning, and perhaps extract simple rules from them so that we can ultimately omit the learning stage entirely. Therefore, we use a decision tree algorithm, because its output is simple and follows transparent logic. Once we better understand the feature space, we can explore more sophisticated algorithms.

Decision trees are built iteratively. At each step, the tree chooses a feature and "splits" on it, resulting in a binary classification. Each feature is ranked by the information gain (percentage of results correctly predicted) if the tree were to split on that feature, and the feature with the highest information gain is chosen. Each split of the tree is then assigned a prediction based on the majority of cases represented within that split. If all cases are correctly predicted, or no further attributes can distinguish the cases, the algorithm terminates. This ultimately produces a decision tree structure that can be translated into a disjunction of hypothesis for the classification problem.

There are several well-known problems associated with decision trees, which we use standard methods to alleviate. To avoid overfitting the tree to the data set, we set a lower bound on information gain in order to prune the tree of splits that add little overall accuracy. We divide our data set into ten parts, and cross-validate our predictions by using nine parts to train the trees and testing on the tenth. We also use k-means clustering techniques to collapse real-valued features into sensible groups. Most of our data analysis occurs within the RapidMiner [3] toolset, using the C4.5 decision tree algorithm.

# 3 Evaluation

Evaluation of our work encounters several challenges. First, success is hard to define, because semantic attributes are usually subjective. For example, if our groupings correspond to "origin", who defines the meaning of origin, or what it means for two files to have "similar origin"? Therefore, we have chosen to evaluate our accuracy against a fairly ubiquitous and descriptive semantic attribute: file extensions.

In practice, predicting file extensions has few applications; most files already have them, and if not, utilities such as `file` can determine them. However, for the purposes of our initial evaluation they are attractive for two reasons. First, they are ubiquitous and undisputed: we do not have to hand-label our training data or make subjective judgements of semantic meaning. In particular, this allows us to easily experiment with large or different data sets.

Second, although we expect that provenance will predict semantic attributes, we do not know precisely what attributes provenance *should* predict. For example, provenance may not be a good predictor of "importance". We need some direction to guide our intuitions. In previous work Mesnier et al. [2] found that file attributes, including extension, predicted future behavior of the file, and extracted predictive rules to tune file system performance. This suggests that the reverse – using file behavior to predict extensions – is worthy of investigation.

Another challenge for our evaluation is finding an appropriate workflow. Although a number of PASSv2 traces have been made available, they all correspond to homogeneous workflows in which a single high-level task was executed. While these workflows are rich in provenance, they do not correspond well to typical heterogeneous user behavior. Ideally, we would construct a comprehensive workflow featuring several different representative applications in tandem. This is a large task on which we are still working. In the interim, we have processed the PASSv2 Linux kernel v2.6.19.1 compile, a data set previously used in other PASSv2 analyses [4].

PASSv2 can only track provenance in directories that have been specially mounted as "Lasagna" volumes, and provenance cannot be shared between Lasagna volumes. Furthermore, the system root cannot be mounted as a Lasagna volume. Consequently, we only track provenance in the compile's directory tree; files outside of the tree appear in the provenance, but do not have their own

provenance tracked. The mounting process introduces a layer of `cp` provenance events, which add some noise.

We only `stat` files that are present on-disk at the end of the workflow. However, workflows also contain temporary files and processes. Consequently, we draw a critical distinction between *manifest* files (which are on-disk and can be `stat`) and non-manifest objects.

We built and installed the PASSv2 kernel v2.6.23.17 on an Intel Core2 Duo running Ubuntu 9.04. We implemented our techniques using python v2.6 and RapidMiner Community Edition, using brute force to compute transitive closures. Recall that we divide the data into ten random parts, train on nine, and test on the tenth.

### 3.1 Results

The Linux kernel compile is a large provenance graph containing 138,243 nodes and 1,338,134 edges. Deversioning reduces it to 68,312 nodes, and name-merging to 34,347 nodes; 21,650 of these are manifest files that we can `stat` after compilation. From the manifest files, we discarded 200 whose extensions appear eight times or less; in practice these do not affect our accuracy, but their removal makes statistical result analysis more tractable. Our current brute-force pipeline takes about a day to process one representation of the graph, so we were only able to compute the following features from merged-name ancestor and descendant subgraphs:

- Node Count
- Edge Count
- Maximum Path Depth
- Neighbors
- File Count
- Process Count

In total, we used 12 provenance features and 11 `stat` features for classification.

Using these 23 features we initially achieved $79.78\% \pm 0.98\%$ extension prediction accuracy on manifest files (see table 2); we need a little over 5,000 training examples for this accuracy to stabilize (see figure 1). On non-manifest files, using just the 12 provenance features we achieved $93.76\% \pm 1.27\%$ accuracy. By merging these two decision trees with a top-level decision between manifest and non-manifest, we achieve 85.68% accuracy across the whole set.

In order to tease apart the classifier's behavior, we ran both manifest and non-manifest files together using only provenance features and achieved $65.5\% \pm 0.23\%$ accuracy. In addition,running just the 11 `stat` features on manifest files achieved $76.05\% \pm 1.03\%$ accuracy

| ext | # in set | precision | recall |
|---|---|---|---|
| .h | 8678 | 96.70% | 72.65% |
| .c | 8420 | 70.22% | 96.94% |
| none | 1869 | 80.26% | 53.08% |
| .S | 912 | 69.34% | 27.52% |
| .o | 829 | 99.28% | 99.76% |
| .txt | 415 | 59.39% | 99.04% |
| .cmd | 147 | 97.24% | 95.92% |
| other | 180 | 31.89% | 15.00% |
| total | 21450 | 82.55% | 79.79% |
| .h+.c+.S | 18010 | 98.76% | 96.10% |
| total | 21450 | 95.83% | 91.87% |

Table 2: Results of the Linux kernel compile analysis on manifest files using provenance and `stat` features.

on manifest objects. We observed in the provenance-only run that most of the classifier's inaccuracy on manifest files came from its inability to distinguish between `.c`, `.h`, and `.S` files. These "source files" are sufficiently similar in usage as to be indistinguishable given the provenance features we collected. If we relaxed our constraints and grouped these extensions into one source file class, then our original provenance-and-`stat` manifest tree would achieve 94.08% accuracy (93.94% across the whole set). However, these source files would then constitute 83.96% of the manifest data set (or 52.44% of the total data set), which makes classification somewhat less challenging.

These results have many interesting implications. First, provenance features classified non-manifest objects exceptionally well. Notably powerful features included ancestral file count, ancestral maximum path depth, and both ancestral and descendant edge counts. The impressive accuracy of provenance features in this case was an unexpected and pleasing result.

Conversely, provenance features alone performed poorly on manifest files, and with the addition of

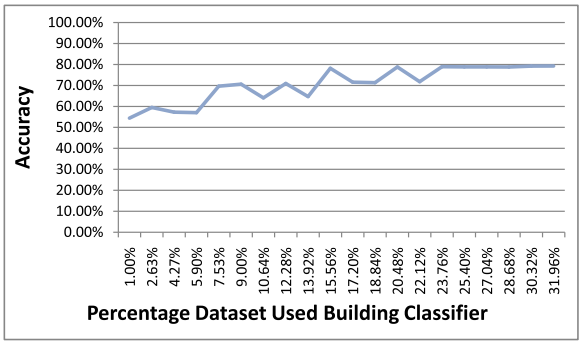


Figure 1: % of Linux Dataset Used vs. Accuracy

`stat` were still substantially harder to predict than non-manifest objects. This was primarily due to the confusion of `.c`, `.h`, and `.S` files. However, this may not be entirely erroneous behavior. As stated previously, it is not necessarily the case that provenance *should* predict file "types" at precisely the semantic level we have chosen. The classifier's error is that it correctly recognizes `.c`, `.h`, and `.S` files as similar in provenance; they are all source files with respect to the compiler. If the graph contained provenance from within the compiler, then these files would be more distinguishable.

However, there does exist information in the original provenance graph that would allow us to distinguish between source files. `.c` files have precisely one `.o` file descendant, whereas `.h` files may have multiple such descendants (and we achieve 99.4% accuracy on `.o` files). Our current feature analysis does not capture this sort of information, and it is not immediately obvious how to do so in a small number of features. One idea is to statistically count the different extensions that appear in the file's ancestors and descendants. Note that we do not suggest "cheating" by using extensions to predict extensions; rather, we suggest that a file's type can be predicted from the types of files it interacts with.

## 4 Future Work

Our results showed a clear predictive link between provenance features and file extension in the Linux kernel compile. This is especially true of non-manifest file system objects, on which we achieved impressive accuracy. We are not yet sure why manifest and non-manifest objects function differently, or if file extensions are at the right semantic level for file "type" prediction. Both of these are interesting topics for future research.

However, our immediate future work is to build a better evaluation data set, as discussed in section 3. While many individual, homogeneous workflows are publically available, there is no representative, heterogeneous provenance data set integrating many workflows over time. We hope that the construction of such a data set will itself be a valuable contribution to future research.

We also have further exploration to do with regards to the feature space. Although we achieved many good and interesting results with a relatively small number of features, many of our other features are not yet tested, and many further avenues of research exist. Edge timestamps are a noteworthy feature that have been successfully used in prior work in desktop search. Our results also indicate that statistical counts of ancestor's and descendant's metadata may be worthy of investigation.

Our current implementation is brute-force and consequently slow on large graphs. Analysis of the Linux compile took place over about a day. As a proof-of-concept this functions, but for our future work we anticipate a second pipeline with more algorithmic finesse. Ultimately, direct integration with the PASSv2 collection layer would allow us to gather many statistics at runtime or in PASSv2's post-processing stages. A production system would probably take the same approach: integration with the file system to gather a few quality features.

## 5 Conclusion

The principal barrier to the widespread adoption of semantic technologies is the difficulty of obtaining rich, descriptive semantic attributes. Existing research into attribute extraction is content-specific and does not consider the historical context of files. We perform machine learning classification on file provenance to predict semantic attributes, in particular file extensions. Because there is a mismatch between the dimensionality and size of provenance vis-a-vis typical feature vectors, we must perform a significant amount of reduction.

Our results show a predictive link between provenance and file extensions, in particular with non-manifest objects. This compliments previous work, which has shown that file extensions and other metadata predict access patterns [2]. We are still working on building the right workflow, refining our choice of features, and improving the processing pipeline's efficiency.

## References


[1] CUI, B., LIU, L., PU, C., SHEN, J., AND TAN, K. Quest: Querying music databases by acoustic and textual features. In *15th International Conference on Multimedia (MULTIMEDIA'07)* (New York, New York, September 2007), ACM.

[2] MESNIER, M., THERESKA, E., ELLARD, D., GANGER, G. R., AND SELTZER, M. File classification in self-* storage systems. In *In Proceedings of the First International Conference on Autonomic Computing (ICAC-04* (2004), pp. 44–51.

[3] MIERSWA, I., WURST, M., KLINKENBERG, R., SCHOLZ, M., AND EULER, T. Yale: Rapid prototyping for complex data mining tasks. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, August 2006), L. Ungar, M. Craven, D. Gunopulos, and T. Eliassi-Rad, Eds., ACM, pp. 935–940.

[4] MUNISWAMY-REDDY, K., BRAUN, U., HOLLAND, D., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. In *2009 USENIX Annual Technical Conference (USENIX'09)* (Berkeley, California, June 2009), USENIX Association.

[5] SHAH, S., SOULES, C., GANGER, G., AND NOBLE, B. Using provenance to aid in personal file search. In *USENIX Annual Technical Conference* (Berkeley, California, June 2007), USENIX Association.

[6] SOULES, C., AND GANGER, G. Towards automatic context-based attribute assignment for semantic file systems. Technical Report CMU-PDL-04-105, Carnegie Mellon University, June 2004.

[7] WANG, G., LU, H., YU, G., AND BAO, Y. Managing very large document collections using semantics. *Journal of Computer Science and Technology 18*, 3 (May 2003), 403–406.